Développement avec Openembedded

Table des matières

Environnement de développement pour Zaurus 1
Prérequis: 1
Installation de l'environnement de développement: 2
Création de packages et/ou d'images (bootstrap, OPIE ou GPE) 4
Débug 6
Ajouter un package à une image 7
Montage et modification de l'image 7
Personnalisation du noyau 8
Pourquoi recompiler notre noyau ? 8
Compilation du noyau 9
Configuration d'un nouveau noyau 9
Personnalisation de GPE 11
Préparation: 11
Noyau 12
Audio/Video 13
Bureautique 13
Réseau 13
Localisation 13
Modification de matchbox-common 14
Divers 14
gpe-fr-image.bb 14
Reconstruction de l'image GPE 14
Toutes les modifications 15
Annexe: Démarrage de GPE 15
buildoz.sh & oz 16
Openembedded & qemu 16
Liens divers 20

Environnement de développement pour Zaurus

Ce chapitre présente le processus permettant la compilation d'applications/images pour Zaurus (C1000) avec openembedded Il est largement inspiré de http://www.openembedded.org/wiki/GettingStarted Cet environnement nous permettra aussi la construction des fameux fichiers 'zImage.bin', 'initrd.bin' et 'updater.sh' (OPIE et/ou GPE) et même d'un feed.

Prérequis:

- Prévoir au minimum 5Go pour la compilation d'un système comme GPE ou OPIE.

- Voir http://www.openembedded.org/wiki/RequiredSoftware et http://www.openembedded.org/wiki/OEandYourDistro

```
apt-get install python2.4 python2.4-dev python2.4-psyco ccache patch quilt m4 sed
               bison make subversion libc6-dev libboost* texi2html
```

Installation de l'environnement de développement:

Installation de bitbake:

Bitbake est un outil de gestion de packages qui permet :

- la cross-compilation
- Les dépendances interpackages
- la construction de packages et ensemble de packages (téléchargement des sources, configuration,compilation et packaging)
- une personnalisation simple

```
mkdir -p /stuff/build/conf /stuff/tools/
cd /stuff/tools
svn co svn://svn.berlios.de/bitbake/branches/bitbake-1.4 bitbake
cd bitbake
./setup.py build
```

Pour plus d'info sur bitbake : cd /stuff/tools/bitbake/doc/manual && make (ou BitBake User Manual http://bitbake.berlios.de/manual/)

Installation des utilitaires ipkg (optionnel):

```
cd /stuff/tools
wget http://handhelds.org/download/packages/ipkg-utils/ipkg-utils-050831.tar.gz
tar xvjf ipkg-utils-050831.tar.gz
mv ipkg-utils-050831 ipkg-utils
```

Installation de monotone

Monotone est un outil de versioning qui va nous permettre d'extraire une branche de la base Openembedded.

```
apt-get install monotone
```

Pour une autre distribution voir http://www.venge.net/monotone/

Installation d'Openembedded

Dans cet exemple nous utiliserons la branche .org.openembedded.oz354X (voir hwr à http://www.oesf.org/forums/index.php?showtopic=20111)

```
cd /stuff
wget http://openembedded.org/snapshots/OE.mtn.bz2
bunzip2 OE.mtn.bz2
# Pour lister les branches de la base
mtn --db=/stuff/OE.mtn list branches
org.openembedded.dev
org.openembedded.documentation
org.openembedded.dreambox
org.openembedded.oz354fam083
org.openembedded.oz354x
org.openembedded.packaged-staging

# Extraction de la branche org.openembedded.oz354x
mtn co --db=/stuff/OE.mtn --branch org.openembedded.oz354x ( ou org.openembedded.d
```

Mise à jour de la base Oe.db:

```
mtn --db=/stuff/OE.mtn pull monotone.openembedded.org org.openembedded.oz354x
```

Mise à jour de la branche:

```
cd /stuff/org.openembedded.oz354x && mtn update
```

Configuration:

```
cd /stuff
cp org.openembedded.oz354x/conf/local.conf.openzaurus-3.5.4 build/conf/local.conf
```

Modifier le fichier build/conf/local.conf de manière à avoir:

```
MACHINE = "akita"
TARGET_ARCH = "arm"
TARGET_OS = "linux"

#adapt these to match your directory layout
DL_DIR   = "/stuff/sources/"
BBFILES := "/stuff/org.openembedded.oz354x/packages/*/*.bb"

#set distro and version
DISTRO = "openzaurus-3.5.4"

#what kind of images do we want?
IMAGE_FSTYPE="jffs2 tar"

#Make use of my SMP box
PARALLEL_MAKE="-j2"

#multimachine build stuff
```

```
KERNEL_STAGING = "${STAGING_DIR}/${PACKAGE_ARCH}-${HOST_OS}/kernel"
STAGING_KERNEL_DIR = "${STAGING_DIR}/${PACKAGE_ARCH}-${HOST_OS}/kernel"
STAMP = "${TMPDIR}/stamps/${PACKAGE_ARCH}-${HOST_OS}/${PF}"
WORKDIR = "${TMPDIR}/work/${PACKAGE_ARCH}-${HOST_OS}/${PF}"

#Add verbosity to make fixing easier
BBINCLUDELOGS = "yes"

#The name says it all
CVS_TARBALL_STASH = "http://www.oesources.org/source/current/"
```

Initialisation de l'environement:

```
export LANG=C
export BBPATH=/stuff/build:/stuff/org.openembedded.oz354x
export PATH=/stuff/tools/bitbake/bin:/stuff/tools/ipkg-utils:$PATH
```

Création de packages et/ou d'images (bootstrap, OPIE ou GPE)

Créer un simple package:

```
bitbake nano
...
Packaged contents of nano into /stuff/tmp/deploy/ipk/nano_1.3.9-r0_armv5te.ipk
Packaged contents of nano-doc into /stuff/tmp/deploy/ipk/nano-doc_1.3.9-r0_armv5te
...
Packaged contents of nano-locale-fr into /stuff/tmp/deploy/ipk/nano-locale-fr_1.3.
```

Bitbake lors de sa première utilisation créeera un cache dans /stuff/tmp/cache/bb_cache.dat. Par la suite ce sera ce cache qui sera utilisé.

Il faut noter que la commande bitbake nano est un ensemble de commande paramétré par le fichier nano….bb qui peut être décomposé comme ceci:

fecth : Telecharge les sources

unpack : décompression des sources

patch : application du / des patch(s)

configure : configuration des sources

compile : compilation

populate_staging : intégration des fichiers compilés

package : création du fichier ipk

donc bitbake nano est en fait la suite de commandes:

```
bitbake -b /stuff/org.openembedded.oz354x/packages/nano/nano_1.3.9.bb -c fectch
bitbake -b /stuff/org.openembedded.oz354x/packages/nano/nano_1.3.9.bb -c unpack
```

```
bitbake -b /stuff/org.openembedded.oz354x/packages/nano/nano_1.3.9.bb -c patch
bitbake -b /stuff/org.openembedded.oz354x/packages/nano/nano_1.3.9.bb -c configure
bitbake -b /stuff/org.openembedded.oz354x/packages/nano/nano_1.3.9.bb -c compile
bitbake -b /stuff/org.openembedded.oz354x/packages/nano/nano_1.3.9.bb -c populate_
bitbake -b /stuff/org.openembedded.oz354x/packages/nano/nano_1.3.9.bb -c package
```

Pratique pour dépolluer certains pb, notamment en utilisant l'option -D (DEBUG)

Quelques commandes '**bitbake**':

La commande suivante pertmet de lister toutes les taches lors de la création d'un package:

```
bitbake -b <path to bb file> -c listtasks
```

Pour consulter toutes les variables utilisées lors de la compilation (ex: nano) :

```
bitbake -e /stuff/org.openembedded.oz354x/packages/nano/nano_1.3.9.bb |more
```

Pour voir la valeur d'une variable OE d'un package:

```
bitbake -b ../openembedded/packages/meta/bootstrap-image.bb -c showdata
```

Important

Special note for task-bootstrap: you may need additional setup for building this very one task.

```
bitbake task-bootstrap
```

Créer un ensemble de packages:

```
cd /stuff
bitbake bootstrap-image
  ...
ls -l tmp/deploy/images/3.5.4.1-rc4/akita/
-rw-r--r-- 1 root root 7110672 2006-05-11 00:47 bootstrap-image-akita-200605102245
-rw-r--r-- 1 root root 4846056 2006-05-11 00:47 bootstrap-image-akita-200605102245
-rw-r--r-- 1 root root 2784412 2006-05-11 00:03 modules-2.6.16-akita.tgz
-rwxr-xr-x 1 root root    5903 2006-05-11 00:35 updater.sh.akita
-rw-r--r-- 1 root root 1141124 2006-05-11 00:03 zImage-2.6.16-akita-20060510191839
```

Créer un groupe de packages:

```
bitbake gpe-image
```

Pour la compilation prévoir au moins 300Mo disponibles pour le téléchargement des sources (DL_DIR) et 4.5Go pour l'espace de travail (/stuff). Après quelques heures de compilation … (et divers petits problèmes avec la version de developpement)…

```
# ls /stuff/tmp/deploy/images/3.5.4.1-rc4/akita/
gpe-image-akita-20060515185250.rootfs.img
gpe-image-akita-20060515185250.rootfs.tar.bz2
modules-2.6.16-akita.tgz
updater.sh.akita
zImage-2.6.16-akita-20060514140125.bin
```

Yes :) De la même manière il est possible de créer une image Opie

```
bitbake opie-image
```

```
ls /stuff/tmp/deploy/images/3.5.4.1-rc4/akita/opie*
opie-image-akita-20060518143804.rootfs.img
opie-image-akita-20060518143804.rootfs.tar.bz2
```

Re yes :) Créer la totalité des packages (déconseillé)

```
bitbake world
```

Pour nettoyer/supprimer un package:

```
bitbake -c clean  LePackage
```

Débug

Lorsqu'un package refuse obstinément de compiler il est souvent intéressant de reprendre les étapes de la compilation une à une.

Pour mémoire, on peut connaitre la série d'actions exécuté sur le package avec la commande:

```
bitbake -b <path to bb file> -c listtasks
```

et les variables positionnées :

```
bitbake -e <path to bb file>
```

on peut ainsi réxécuter les actions

```
...
bitbake -b <path to bb file> -c configure
bitbake -b <path to bb file> -c compile
...
```

Ce peut ne pas être suffisant, les options de configuration et de compilation étant fixées dans le fichier bb peuvent surement être affinées (???)

Openembedded lorsqu'il exécute une actions (fetch, configure, compile, ...) créée en premier lieu le script shell qui représente cette action et l'exécute tout en loguant le résultat. Intéressant ... Il est donc possible de rééxécuter un script ayant échoué ? OUI ;-)

Tout dabord on se rend à l'emplacement des sources du package. Imaginons que le package 'openssl' ce soit planté lors de l'éxécution du **do_compile**. (ou tout autre actions).

```
cd /stuff/dev/tmp/work/armv5te-linux/openssl-0.9.7g-r2
ls temp
run.do_configure.3581
log.do_configure.3581
run.do_compile.3581
log.do_compile.3581
```

Copions le fichier `run.do_compile.3581` en debug.sh. Dans ce dernier recherchons la fonction 'do_compile' (ou do_confiure, ou ...) et remplacons oe_make par bash. Lors de l'éxécution de ce script nous récupérons un shell avec l'environnement de cross compilation correctement paramétré.

Et donc maintenant un simple make reproduit l'erreur constatée. Il ne reste plus qu'à en connaitre la cause ;)

Ajouter un package à une image

Il nous est aussi possible d'ajouter un package à une image.

Imaginons que nous souhaitions que l'editeur 'nano' soit intégré à notre image OPIE.

Nous ajoutons simplement ce package au fichier à la variable **INSTALL_PACKAGES** du fichier `/stuff/org.openembedded.dev/packages/meta/opie-image.bb` qui devient:

```
INSTALL_PACKAGES = "task-bootstrap task-opie-base task-opie-base-applets \
                    task-opie-base-inputmethods task-opie-base-apps \
                    task-opie-base-settings task-opie-base-decorations \
                    task-opie-base-styles task-opie-base-pim \
                    task-opie-extra-settings \
                    task-opie-bluetooth task-opie-irda \
                    nano \
                    ${extra_stuff}"
```

Et on relance

```
bitbake opie-image
```

Peut pas faire plus simple :)

Cette méthode nous permet de personnaliser très facilement une image openzaurus. Nota:

Les fichiers BB permettent la construction des packages, on y trouve divers dépendances:

DEPENDS : sont les packages dont nous auront besoin pour la compilation

RDEPENDS : sont ceux dont nous auront besoin pour l'exécution.

Montage et modification de l'image

Voyons voir ce que contient notre image.

Pour modifier manuellement notre image (par ex gpe-image.img) il nous faut la ‘monter’. (voir le site d’Ulhume qui explique très bien cela : http://artisan.karma-lab.net/node/59)

Pré requis: mtd-tools

```
apt-get install mtd-tools
```

```
cp /stuff/tmp/deploy/images/.../gpe-image-akita-......rootfs.img /tmp/gpe.img
modprobe mtdblock
mknod /dev/openzaurus b 31 0
modprobe mtdram total_size=65535
dd if=/tmp/gpe.img of=/dev/openzaurus bs=16 skip=1
mkdir /tmp/openzaurus
mount -t jffs2 /dev/openzaurus /tmp/openzaurus
ls /tmp/openzaurus
```

On peut maintenant personnaliser notre image. Et enfin construire une nouvelle image y incluant nos modifications.

(voir `/stuff/org.openembedded.dev/conf/bitbake.conf`)

```
mkfs.jffs2 --root=/tmp/openzaurus/ --faketime --little-endian --eraseblock=0x4000
cat /stuff/tmp/staging/arm-linux/lib/sharp-flash-header/header-c700.bin /tmp/my-gp
```

flashage de /tmp/my-gpe.img (en initrd.bin) Et pour finir un peu de nettoyage.

```
umount /tmp/openzaurus && rmdir /tmp/openzaurus
rmmod jffs2 mtdram mtdblock
rm -f /dev/openzaurus
```

Personnalisation du noyau

Ce document décrit une méthodologie permettant la compilation et l’installation d’un nouveau noyau Zaurus C1000. Dans le cas précis j’ajoute les modules permettant le firewalling et la gestion de la Qos.

Pourquoi recompiler notre noyau ?

Le noyau de base installé sur notre Zaurus supporte diverses fonctionnalités qui sont directement intégrées dans le noyau ou sous forme de module. Dans la majorité des cas ce noyau est suffisant pour une exploitation normal de l’appareil et à la reconnaissance de la plupart des matériels.

Mais imaginons que nous ayons reçu un nouveau matériel qui ne serait pas reconnu ou que nous souhaitions intégrer une fonction qui ne serait pas implémentée. C’est alors qui nous faudrai recompiler le noyau pour cette prise en charge.

En fait j’ai fais cette doc, car les modules relatifs à la gestion d’un firewall n’était pas intégrés dans le noyau fourni en standard. Bon dacoord cela n’est pas vital à l’exploitaion d’un Zaurus mais pourquoi s’en priver. La machine pour laquelle je recompile le noyau est une SLC-1000 (akita), la méthode reste la même pour un autre type de machine (voir `/stuff/org.openembedded.oz354x/conf/machine/`)

Compilation du noyau

Pour permettre la compilation d'un noyau Zaurus sous X86, il est necessaire d'installer l'environnement de cross-compilation comme indiqué dans ce document: Environnement de développement pour Zaurus Cette première étape ayant été franchie, nous allons maintenant tester notre environnement. Commencons simplement par la compilation d'une image zImge.bin et initrd.bin minimum (bootstrap). Nous commencons par charger le module jffs2 pour la création de l'image initrd.bin

```
(modprobe jffs2)
bitbake task-bootstrap
bitbake bootstrap-image
```

Des dépendances sont tout dabord compilées (ipkg-utils-native, file-native, flex-native, bison-native, binutils-cross, gcc-cross-initial, linux-libc-headers, glibc, gmp-native, mpfr-native, gcc-cross) puis débute la création des packages ipk.

Nota: Les sources du noyau sont décompressées dans `/stuff/tmp/work/akita-linux/linux-openzaurus-2.6.16-r??/linux-2.6.16/`, une série de patchs est appliquée, le fichier de config est copié à la racine des sources et la compilation débute … La configuration du noyau standard est présente dans le fichier (selon la machine utilisée): `/stuff/org.openembedded.dev/packages/linux/linux-openzaurus-2.6.16/defconfig-akita` A l'issue de la compilation le noyau zImage… est présent dans `/stuff/tmp/deploy/images/` , des packages ont été crées dans `/stuff/deploy/ipkg`

Configuration d'un nouveau noyau

Pour éviter de modifier manuellement le fichier de configuration nous utiliserons le script make des sources du noyau. Tout dabord sauvegarde de la config de base

```
cp /stuff/org.openembedded.oz354x/packages/linux/linux-openzaurus-2.6.16/defconfig
   /stuff/org.openembedded.oz354x/packages/linux/linux-openzaurus-2.6.16/defconfig
```

Les divers menus nous permettent de customiser notre noyau.

J'ajoute la gestion iptables pour la prise en charge du firewalling (Dans Networking/Networking options/Network packet filtering/Core Netfilter Configuration et Netfilter Configuration). Pour les courageux il est aussi possible d'ajouter la gestion de la QoS.

Ensuite je sauvegarde et sors de l'utilitaire de configuration.

Le fichier `.config` est présent avec toutes nos modifications, c'est donc celui-ci qui sera utilisé lors de la compilation du noyau. Pour cela on remplace la config du package linux-openzaurus.

```
cp .config /stuff/org.openembedded.oz354x/packages/linux/linux-openzaurus-2.6.16/d
```

La config de notre noyau étant terminée, comment pouvons nous indiqué à bitbake qu'il lui faut intégrer les nouveaux modules ?

Voyons voir le fichier qui permet cette compilation de l'image minimal. Et comme les choses sont bien faites, il se nomme `bootstrap-image.bb`

```
vi /stuff/org.openembedded.oz354x/packages/meta/bootstrap-image.bb
```

Pas bavard :(par contre on apprend qu'il depend de **task-bootstrap**

Lecture de celui-ci et la encore des dépendances.

```
vi /stuff/org.openembedded.oz354x/packages/meta/task-bootstrap.bb
...
RDEPENDS = 'base-files base-passwd busybox \
        initscripts \
        netbase sysvinit sysvinit-pidof tinylogin \
        modutils-initscripts fuser setserial\
        ${HOTPLUG} \
        ${BOOTSTRAP_EXTRA_RDEPENDS} \
        ${@bootstrap_modutils_rdepends(d)}'
```

L'une d'elle nous intéresse plus particulièrement : **BOOTSTRAP_EXTRA_RDEPENDS**

Bon très bien mais comment je fais pour savoir d'ou elle est héritée ? On cherche :)

```
find /stuff/org.openembedded.oz354x | xargs grep BOOTSTRAP_EXTRA_RDEPENDS
```

oh oh apparement ces dépendances sont fournies par dans les fichiers de config relatifs au type de machines c'est à dire `/stuff/org.openembedded.oz354x/conf/machine/*`

et moi j'ai un akita

J'ouvre donc le fichier `/stuff/org.openembedded.oz354x/conf/machine/akita.conf`

Aie pas de **BOOTSTRAP_EXTRA_RDEPENDS** par contre ce fichier hérite aussi de `conf/machine/zaurus-clamshell-2.6.conf` Dans celui-ci on y trouve enfin notre, ou plus exactement nos **BOOTSTRAP_EXTRA_RDEPENDS**

C'est donc dans celui-ci que nous pourrons ajouter nos nouveau modules.

Mais avant de les ajouter et nous faut en encore leurs noms exact.

Il y a peut être plus simple mais moi je compile un linux-openzaurus qui me générera les kernel-modules-* Puisque j'ai déjà fais une compilation du noyau je nettoie les sources pour la prise en compte de nos nouvelles options du kernel.

```
cd /stuff
ls /stuff/tmp/deploy/ipk/kernel* > avant.txt
rm /stuff/tmp/deploy/ipk/kernel*
bitbake -c clean linux-openzaurus
bitbake linux-openzaurus
ls /stuff/tmp/deploy/ipk/kernel* > apres.txt
diff avant.txt apres.txt
```

Voilà nous avons maintenant la liste des modules qui devront être intégrés à notre image.

Je les ajoute au fichier `/stuff/org.openembedded.oz354x/conf/machine/zaurus-clamshell-2.6.conf`

Bon certes j'ai été un peu gourmand :) Création de notre image bootstrap:

```
cd /stuff
rm tmp/deploy/images/*
bitbake -c clean bootstrap-image
```

```
bitbake -c clean linux-openzaurus
bitbake -c clean zaurus-updater
bitbake bootstrap-image
```

Nous pouvons maintenant de tester notre noyau en bootant directement sur l'image bootstrap que nous venons de créer.

Nota: faire un 'Fn' + 'flechedroite' pour avoir un prompt

Et un modprobe ip_tables nous confirme l'installation des nouveaux modules :) Et maintenant notre image GPE avec un kernel supportant Netfilter

Personnalisation de GPE

L'objectif de ce tutoriel étant donner un mode d'emploi permettant le personnalisation et la construction d'une image GPE française.

Notre image devra:

1 - Noyau - source 2.6.17

- inclure les modules iptables de Netfilter et gestion de la Qos
- inclure le module Frame Bruffer pour w100
- modification du logo

2 - Audio/Video: + mplayer + vlc-gpe

3 - Bureautique + Abiword + vim

4 - Réseaux + tcpdump + nmap + kismet

5 - Localisation + française

6 - Personnalisation de matchbox

7 - Divers + firefox 1.5 + gpe-filemanager

8 - Suppression de programmes -gpe-edit

Préparation:

L'environnement de développement devra être mis en place (voir Environnement de développement pour Zaurus [http://dab.free.fr/www/?/Zaurus/Developpement-avec-Openembedded])

Au cours de cette doc je travaillerai dans l'arborescence de **/stuff/org.openembedded.oz354x_fr** qui sera une copie de /stuff/org.openembedded.oz354x. Je pourrai ainsi créer un fichier `.diff`

```
cp /stuff/org.openembedded.oz354x /stuff/org.openembedded.oz354x_fr
mv /stuff/org.openembedded.oz354x /stuff/org.openembedded.oz354x.orig
```

Noyau

Pour la personnalisation du noyau : voir Personnalisation du noyau [http://dab.free.fr/www/?/Zaurus/Developpement-avec-Openembedded2]

Modification du logo:

Le fichier linux-openzaurus_2.6.17.bb nous apprend que 3 patchs appliqués sont relatifs au logo.

${RPSRC}/logo_rotate_fix-r1.patch

${RPSRC}/logo_oh-r0.patch.bz2

${RPSRC}/logo_oz-r2.patch.bz2

$RPSRC est défini dans linux-openzaurus.inc et est égale a "http://www.rpsys.net/openzaurus/patches/archive [http://www.rpsys.net/openzaurus/patches/archive/]" En regardant le contenu de ces fichiers on peut constater que le patch `logo_oz-r2.patch.bz2` contient les fichiers ppm des logos et que le patch s'applique aux 3 fichiers `drivers/video/logo/logo_oz240_clut224.ppm` / `logo_oz480_clut224.ppm` / `logo_oz640_clut224.ppm` dans les sources du noyau linux. Il nous suffit donc de modifier ces fichiers et de relancer la compilation du noyau. Tout dabord un peu de nettoyage

```
bitbake -c clean linux-openzaurus
rm /stuff/tmp/deploy/ipk/kernel*
```

On debute la compilation et la stopppe apres l'application des patchs

Ensuite nous modifierons les 3 fichiers logos … En fait pour un akita seul le fichier `logo_oz640_clut224.ppm` devra être remplacé.

Creation d'un logo PPM partir d'un image JPEG

```
#! /bin/bash
# Transformer le .jpg en paramètre du script en .pnm
jpegtopnm $1 > tmp.pnm
# Quantifier à 224 couleurs
pnmquant 224 tmp.pnm > tmp2.pnm
# Transformer au format PPM no raw
pnmnoraw tmp2.pnm > logo_oz640_clut224.ppm
rm tmp.pnm tmp2.pnm
```

On relance à nouveau la compilation

```
bitbake linux-openzaurus
```

Et voilà notre nouveau noyau est prêt :)

Nous aurions pu pour le tester créer une image bootstrap.

```
bitbake bootstrap-image
```

Passons maintenant à la personnalisation de GPE.

Une copie du fichier `gpe-image.bb` nous permettra de personnaliser le contenu de notre image.

```
cp /stuff/org.openembedded.oz354x/packages/meta/gpe-image.bb/stuff/org.openembedde
```

Nous modifierons le fichier `/stuff/org.openembedded.oz354x/packages/meta/gpe-fr-image.bb` en y ajoutantjuste avant '**export IPKG_INSTALL**' les divers **GPE_EXTRA_INSTALL**

Audio/Video

Ajout de mplayer

```
bitbake mplayer
bitbake mplayer-common
```

.. compilation et création de toutes les dépendances qui vont bien ...

```
GPE_EXTRA_INSTALL += mplayer
```

Bureautique

```
bitbake abiword
bitbake vim
```

```
GPE_EXTRA_INSTALL += abiword vim
```

J'ai aussi ajouté le fichier `packages/base-files/base-files/share/dot.vimrc` et modifié `packages/base-files_3.0.14.bb` pour avoir la colorisation sous vim.

Réseau

```
bitbake tcpdump
bitbake kismet
bitbake nmap
```

```
GPE_EXTRA_INSTALL += tcpdump kismet nmap
```

Localisation

Le fichier `glibc_2.3.5+cvs20050627.bb` sera modifié de manière a appliquer un patch qui modifiera `libc/localedata/SUPPORTED`. Inutile de compiler les langues qui ne nous intéresse pas. J'ai simplement ajouté la ligne suivante au fichier bb :

```
file://SUPPORTED_fr.patch;patch=1"
```

Le patch `SUPPORTED_fr.patch` sera à copier dans `packages/glibc/glibc-cvs-2.3.5/`

Pour l'instant j'ajoute simplement les paquets localisés.

```
GPE_EXTRA_INSTALL += gpe-aerial-locale-fr gpe-beam-locale-fr gpe-calendar-locale-f
```

Modification de matchbox-common

```
cp /stuff/sources/matchbox-common-*.tar.gz .
tar xvzf matchbox-common-*.tar.gz

# modifications des fichiers situés dans matchbow-common*data/vfolders-pda/
# ...

tar cvzf matchbox-common-0.9.1.tar.gz matchbox-common-0.9.1/
cp matchbox-common-0.9.1.tar.gz /stuff/sources
bitbake -c clean matchbox-common
```

Divers

```
bitbake firefox
bitbake gpe-filemanager


GPE_EXTRA_INSTALL += mplayer abiword vim gpe-filemanager gpe-aerial-locale-fr gpe-
```

gpe-fr-image.bb

Au final notre fichier gpe-fr-image.bb ressemblera à:

```
...
DEPENDS = "task-bootstrap \
           meta-fr-gpe \
           ${GPE_EXTRA_DEPENDS}"


# Audio/Video
GPE_EXTRA_INSTALL += mplayer xmms

# Texte
GPE_EXTRA_INSTALL +=  abiword gnumeric vim gpdf

# Reseau
GPE_EXTRA_INSTALL += tcpdump kismet nmap

# Localisation francaise
GPE_EXTRA_INSTALL += gpe-aerial-locale-fr gpe-beam-locale-fr gpe-calendar-locale-f

# Divers
GPE_EXTRA_INSTALL += firefox gpe-filemanager

export IPKG_INSTALL = "task-bootstrap gpe-task-base \
...
```

Reconstruction de l'image GPE

Puisque nous venons de modifier le fichier `meta-gpe.bb` il nous faut donc purger le package pour permettre sa reconstruction.

```
bitbake -c clean meta-gpe
bitbake gpe-fr-image
```

Toutes les modifications

Il est possible de construire un patch

```
diff -aburN org.openembedded.oz354x.orig/ org.openembedded.oz354x_fr > org.openemb
```

C'est ce que j'ai fais (voir le chapitre suivant :)

Annexe: Démarrage de GPE

http://www.oesf.org/forums/index.php?showtopic=18353&st=45

Ok, here’s breakdown how OZ launches its X11 system:

1. /etc/init.d/gpe-dm launches /etx/X11/Xserver

Xserver is a shellscript which sets mutliple paramters for the real X server, like rotation, DPI and input device and the selects the correct xserver binary as well. For OZ that would be kdrive-fbdev.

2. After the xserver is running, all scripts in /etc/X11/Xinit.d are sourced and executed.

Various things are configured at this stage: correct xmodmap (x11 keymapping), TS calibration if required and the mouse pointer is made invisible.

The last script launches gpe-login, the GPE login manager

3. After gpe-login is used to login as a user, a new xserver running with the permissions of $USER is launched and all scripts in /etc/init.d/Xsession.d are executed

At this stage all user-space stuff required for GPE operation is launched under the UID of $USER

(ipaq-sleep, gconfd, keylaunch etc)

The very last script in Xsession.d launches the GUI:

QUOTE

cat 99xWindowManager

#!/bin/sh

exec x-window-manager

#

x-window-manager is a symlink handled by update-alternatives and points to the session file used to bring up the GUI.

Reroute that script to something else to launch another WM.

The typical GPE session consists of:

matchbox-panel

matchbox-desktop

matchbox-wm

buildoz.sh & oz

Créez et personnalisez simplement vos images, packages et feed

Les commandes suiavntes permettent de créer un environnement Debian, indépendant de la distribution utilisée.

```
wget http://dab.free.fr/buildoz/buildoz.sh
chmod +x buildoz.sh
./buildoz.sh
```

Un 'chroot' vers notre environnement fraichememt construit ...

```
chroot DebianOE /root/oechroot
```

Et voilà nous pouvons maintenant construire nos images.

```
oz build oz354X
```

:))

Openembedded & qemu

Qemu émule diverses actitectures dont une qui nous intéresse plus particulièrement : ARM :)

Il nous est donc possible d'émuler un Zaurus ? OUI :)

Pour écrire cet article je me suis inspiré des scripts fournis par la branche 'Poky' : http://projects.o-hand.com/poky

Pour pouvoir réaliser cela plusieurs prérequis sont nécessaires.

- Un environnement Openembedded en 'bon état de marche' : http://dab.free.fr/www/?/Zaurus/Developpement-avec-Openembedded [http://dab.free.fr/?/Zaurus/Developpement-avec-Openembedded] . Dans la suite de cet article nous supposerons que l'environnement DebianOE est installé dans /mnt

- Diposer de la branche ‘Poky’ : se chrooter dans l’environnement DebianOE et construire la branche poky

```
chroot /mnt/DebianOE /root/oechroot
oz build poky
```

A l’issue de la compilation une image et un noyau sont créés.

```
ls /mnt/DebianOE/stuff/poky/build/tmp/deploy/images/
```

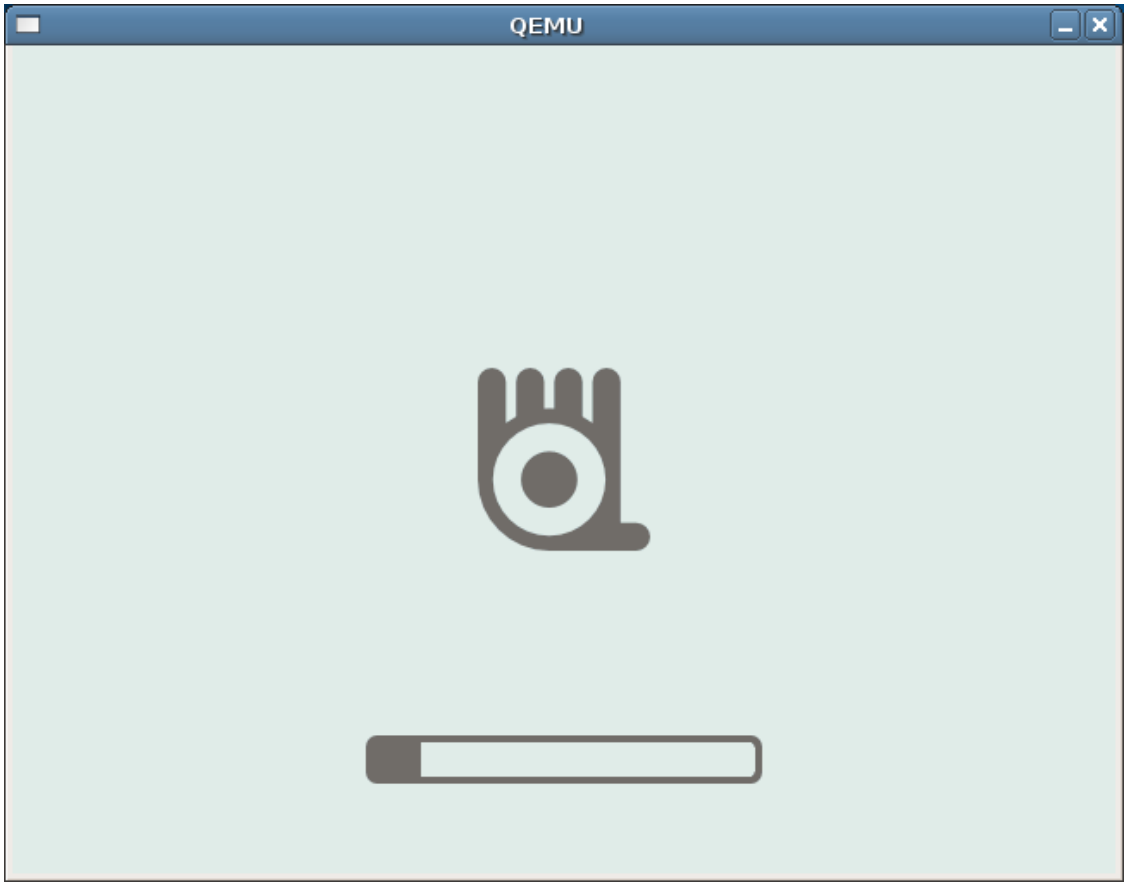
```
modules-2.6.17-qemuarm.tgz
oh-image-pda-qemuarm-20061008213718.rootfs.ext2
oh-image-pda-qemuarm-20061008213718.rootfs.tar.bz2
updater.sh.akita
zImage-2.6.17-qemuarm-20061008161924.bin
```

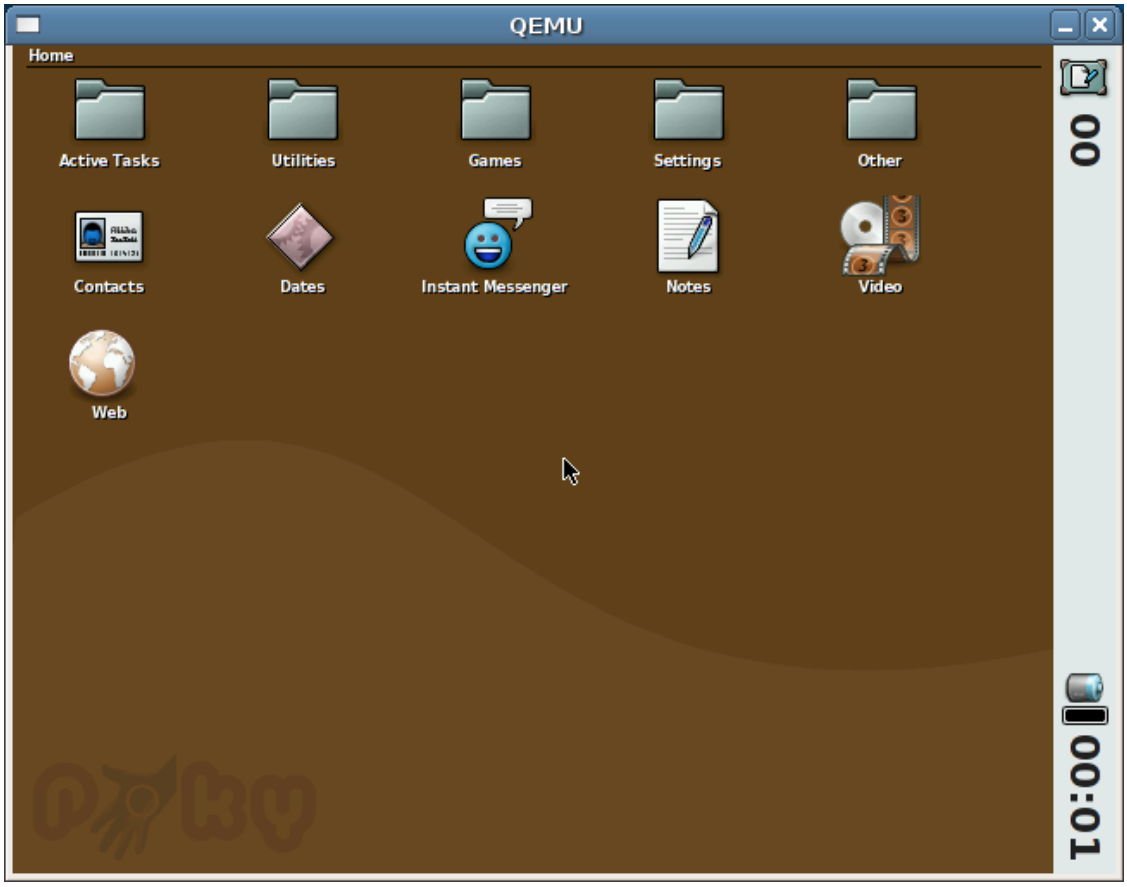
Nous avons maintenant tous les ingrédiants pour admirer le résultat

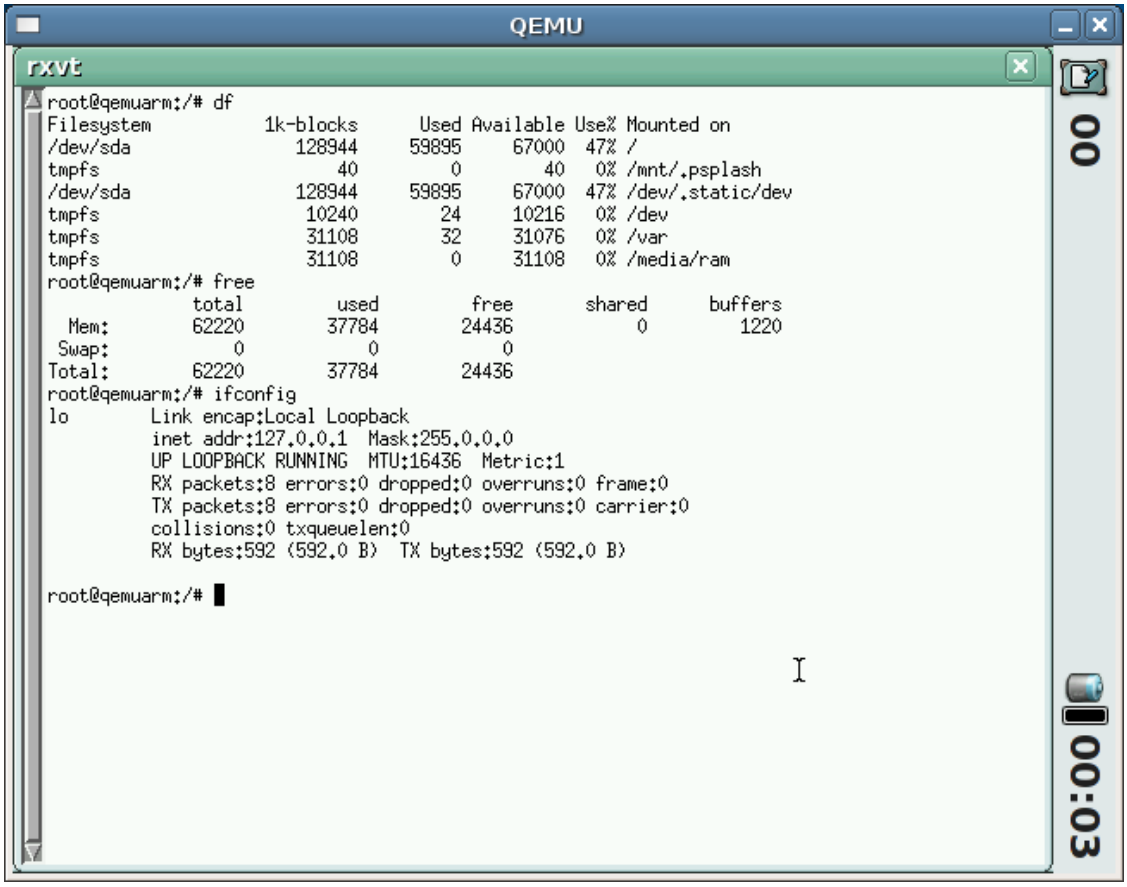
Au passage il faut noter que nous utiliserons l’emulateur qemu-system-arm fourni pas le package qemu-native. (Des patchs sont ajouter pour la prise en compte de l’ecran tactile)

```
qemu-system-arm -kernel zImage-2.6.17-qemuarm-20061008161924.bin -append "root=/de
```

QEMU
smc91x.c: v1.1, sep 22 2004 by Nicolas Pitre <nico@cam.org>
eth0: SMC91C11xFD (rev 1) at c4866000 IRQ 25 [nowait]
eth0: Ethernet addr: 52:54:00:12:34:56
PCI: enabling device 0000:00:0c.0 (0140 -> 0143)
sym0: <895a> rev 0x0 at pci 0000:00:0c.0 irq 27
sym0: No NVRAM, ID 7, Fast-40, LVD, parity checking
sym0: SCSI BUS has been reset.
scsi0 : sym-2.2.3
Vendor: QEMU Model: QEMU HARDDISK Rev: 0.8.
Type: Direct-Access ANSI SCSI revision: 03
target0:0:0: Beginning Domain Validation
target0:0:0: asynchronous
target0:0:0: asynchronous
target0:0:0: Domain Validation skipping write tests
target0:0:0: Ending Domain Validation
Vendor: QEMU Model: QEMU CD-ROM Rev: 0.8.
Type: CD-ROM ANSI SCSI revision: 03
target0:0:2: Beginning Domain Validation
target0:0:2: asynchronous
target0:0:2: asynchronous
target0:0:2: Domain Validation skipping write tests
target0:0:2: Ending Domain Validation


QEMU


QEMU
Home
Active Tasks
Utilities
Games
Settings
Other
Contacts
Dates
Instant Messenger
Notes
Video
Web
00
10:00:01


QEMU
rxvt
root@qemuarm:/# df
Filesystem 1k-blocks Used Available Use% Mounted on
/dev/sda 128944 59895 67000 47% /
tmpfs 40 0 40 0% /mnt/.psplash
/dev/sda 128944 59895 67000 47% /dev/.static/dev
tmpfs 10240 24 10216 0% /dev
tmpfs 31108 32 31076 0% /var
tmpfs 31108 0 31108 0% /media/ram
root@qemuarm:/# free
total used free shared buffers
Mem: 62220 37784 24436 0 1220
Swap: 0 0 0
Total: 62220 37784 24436
root@qemuarm:/# ifconfig
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
UP LOOPBACK RUNNING MTU:16436 Metric:1
RX packets:8 errors:0 dropped:0 overruns:0 frame:0
TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:592 (592.0 B) TX bytes:592 (592.0 B)
root@qemuarm:/#
00
00:03

Il nous manque tout de même l'essentiel : le réseau :(Qemu accéde à la couche réseau par un tunnel créé entre la machine réelle et celle émulée. tout dabord nous allons créer l'interface 'tun0' sur la machine physique. chargeons pour cela le module adéquat.

```
modprobe tun
```

La configuration de l'interface tun0 se fait simplement # ifconfig tun0 10.1.1.1 Bon jusque là tout va bien on va faire la même manip sur la machine virtuelle. (en changeant simplement d'adresse ip)

```
modprobe tun
```

FATAL: Module tun not found. Aie le module n'est pas fourni :(:(… En effet aucun modules n'est présent dans `/lib/modules/2.6.17/`

hmmm ... :(((

Liens divers

Tableau 1.

Indispensable	http://www.openembedded.org/user-manual

Wiki OE	http://openzaurus.berlios.de/ (mirror1) [http://openzaurus.berlios.de/] (mirror2) [http://ozwiki.stuvel.eu/Main_Page]